SPM project: Parallel Prefix

Gaspare Ferraro, 520549

Master Degree in Computer Science - University of Pisa

ferraro@gaspa.re

November 4, 2018

1 Introduction

In this report we will analyze, theoretically and practically, the resolution of the problem of the (parallel) prefix sum:

Given a vector $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ *and a binary operation* $\oplus$*, compute the vector* $y = \langle x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} \rangle$.

In literature this operation is also called (inclusive) scan or partial sum.

For the analysis of the problem we have to make two assumptions:

- The binary operation $\oplus$ is associative ($a \oplus (b \oplus c) = (a \oplus b) \oplus c$) and commutative ($a \oplus b = b \oplus a$), this is an important assumption as we will see in the next chapters the order of the operations may not be preserved.
- The size of the input vector is a power of 2, not a strong assumption, as all the algorithms we will present could be easily generalize to all the sizes, but it only helps to simplify some operations.

2 Sequential algorithm

The sequential algorithm simple compute each element of the vector y as follow:

$$y_i = \begin{cases} x_0 & \text{for } i = 0 \\ x_i \oplus y_{i-1}, & \text{for } 0 < i < n \end{cases}$$

This algorithm is optimal in a sequential model as it has a running time of $\mathcal{O}(n)$, assuming that $\oplus$ is $\mathcal{O}(1)$, and performs $n-1$ calls to $\oplus$ operation.

3 Parallel architecture design

The problem of computing the prefix sum vector is a classical example of a problem that have an optimal solution in a sequential model but that can be optimized in a parallel model. The optimization is not in terms of total complexity or in the number of $\oplus$ operations performed (which are already optimals in the sequential algorithm) but in terms of completion time. When the number of available workers is more than one we can trade-off more total work for less completion time.

We will now introduce two different algorithms that solve in an efficient way the prefix sum problem in a parallel model.

3.1 Block-based algorithm

The idea behind the first parallel algorithm is to compute the prefix vector in three phases:

- Partitionate the input vector in blocks and compute. in parallel, the prefix vector of each of them.
- In the second phase we put the final element of each block in a temporary vector and compute its prefix vector.
- Finally for each block i (starting from the second) add, in parallel, the $(i-1)$th element of the temporary vector.

At the end of the three phases the final vector contains the correct prefixes.

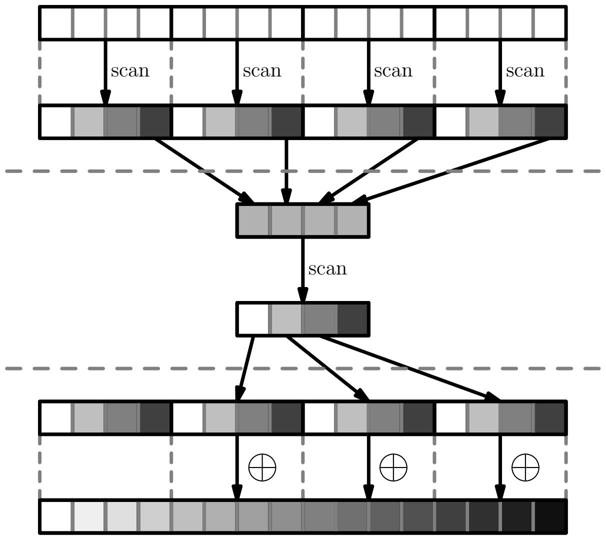


Figure 1: Block-based algorithm graphic representation

3.2 Circuit-based algorithm

Another class of algorithms are the one based on a circuit-like representation. A generic prefix circuit C is a collection of tasks $T_1, T_2, \ldots, T_t,$. A task T_i is a set of operations (l, r) that means to compute $x_r = x_r \oplus x_l$. The tasks must be computed sequentially from 1 to t but the advantage is that all the operations in a same task can be performed in parallel without race conditions.

For example the sequential algorithm seen before can be represented as a prefix circuit $S(n) = \{T_1, T_2, \ldots, T_{n-1}\}$ where $T_i = \{(i-1, i)\}$.

We present now the simple (but still efficient) prefix circuit $P(2^m)$:

$$P(2^m) = \{T_1, T_2, \ldots, T_m, T_{m+1}, \ldots, T_{2m-1}\}$$

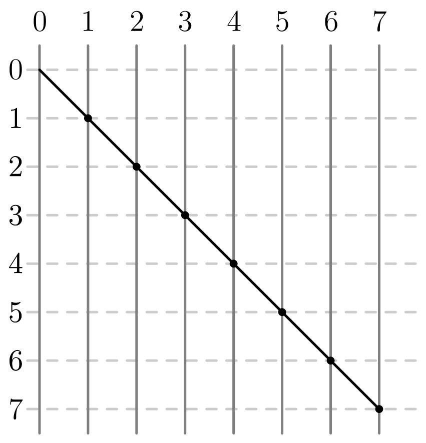
where for $i = 1, \ldots, m$:

$$T_i = \{(2k^i - 2^{i-1} - 1, 2k^i - 1) \mid k = 1, \ldots, 2^{m-i}\}$$

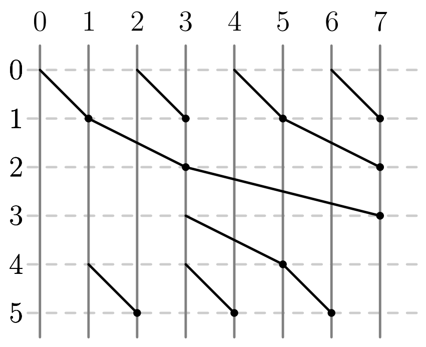
and for $i = 1, \ldots, m-1$:

$$T_{m+i} = \{(k2^{m-i} - 1, k2^{m-i} + 2^{m-i-1} - 1) \mid k = 1, \ldots, 2^i - 1\}$$

In the first phase (from 1 to m) we compute operations like in a reduction tree, in the second phase (from $m+1$ to $2m-1$) we compute the final value of each prefix as the sum of only two others values.



(a) Prefix circuit $S(8)$



(b) Prefix circuit $P(8)$

Figure 2: Examples of prefix circuits

As we can see in these simple examples, for $n = 8$ the sequential algorithm needs 7 units of time against the only 5 of the parallel algorithm (obviously if the number of workers is at least 4).

4 Performance modeling

For the theoretical performance modeling we will denote with:

- $n = 2^m$ the size of the input vector ($m = \log_2 n$).
- p the parallelism degree (for simplicity $1 \leq p \leq n$).
- $t_\oplus$ is the time to compute a single $\oplus$ operation.

We will use this assumption to estimate a theoretical completion time for each algorithm seen.

4.1 Sequential algorithm

The completion time of the sequential algorithm is simply $T_C^s = (n-1) \times t_\oplus$.

4.2 Block-based algorithm

The parallel completion time of the block-based algorithm is the sum of the completion times of all of the three phases:

$$T_{C_p}^b \simeq T_{1_p} + T_{2_p} + T_{3_p}$$

Where:

- $T_{1_p} \simeq (n/p) \times t_\oplus$ is the time to compute the prefix sums of a single block (as all the p blocks are performed in parallel).
- $T_{2_p} \simeq p \times t_\oplus$ is the time to compute the prefix sums over the temporary vector.
- $T_{3_p} \simeq (n/p) \times t_\oplus$ (or 0 if $p = 1$) is the time to add a value to a single block (except the first one).

So the total completion time is $T_{C_p}^b \simeq (2n/p + p) \times t_\oplus$.

In this analysis we assume that n is a multiple of p and all the blocks have the same size, if this is not the case we can consider the block size as the size of the biggest block.

4.3 Circuit-based algorithm

In the prefix circuit $P(n)$ we have a total of $2m-1$ tasks that must be executed sequentially. As said before all the operations in the same task can be performed in parallel at the same time, so a generic task T_i has a completion time of $1 \times t_\oplus$.

So the completion time of the algorithm is $T_{C_p}^c \geq (2\log_2(n) - 1) \times t_\oplus$.

However this analysis works only if the parallelism degree is high enough to allow to performs all the operations in T_i at the same time ($p \geq n/2$).

When this is not the case a task T_i is perfomed in $\max(1, |T_i|/p) \times t_\oplus$ time and the completion time is circa $T_{C_p}^c \simeq 2(\log_2 p + n/p) \times t_\oplus$.

4.4 Comparison table

Now we can compare the expected completion time of all the algorithms:

p	T^s_C	$T^b_{C_p}$	$T^c_{C_p}$
1	$(n-1) \times t_\oplus$	$n \times t_\oplus$	$2n \times t_\oplus$
2	$(n-1) \times t_\oplus$	$n \times t_\oplus$	$(n+2) \times t_\oplus$
4	$(n-1) \times t_\oplus$	$(n/2+4) \times t_\oplus$	$(n/2+4) \times t_\oplus$
8	$(n-1) \times t_\oplus$	$(n/4+8) \times t_\oplus$	$(n/4+3) \times t_\oplus$
...	...	...	...
n/2	$(n-1) \times t_\oplus$	$(n/2+1) \times t_\oplus$	$2(\log_2(n)+1) \times t_\oplus$
n	$(n-1) \times t_\oplus$	$(n+2) \times t_\oplus$	$2(\log_2(n)+1) \times t_\oplus$

Table 1: Comparison of the three algorithms with different parallelism degrees

What we can see from the table is that until $p = 2$ the improvement is negligible, but when the parallelism degree increase the parallel algorithms scales very well.

The circuit-based algorithm reach its best performance for $p \geq n/2$ where each task is computed in $1 \times t_\oplus$. The block-based algorithm scales as well as the circuit-based, but when the parallelism degree increase we have a lack of performance due the trade-off between the parallel part of the first and the third stage ($2n/p \times t_\oplus$) and the second stage that must be performed sequentially ($p \times t_\oplus$).

In practive we expect, for p equals to 1 and 2, the completion time for the parallel algorithms to be similar or greater than the sequential algorithm.

5 Implementations structure and details

All the implementations are written in C++17, the source code is available as attachment with the report or on GitHub. Both of the parallel algorithms have been implemented using the standard C++ threads, the FastFlow framework and the OpenMP/Cilk Plus API. All the algorithms are designed to avoid race conditions between the operations in the same phase, so there is no need for any special synchronization mechanism except for the barriers at the end of each phase to pass to the next one.

Each different implementation is available as single class in a single *.hpp* file. All the classes provide the methods to change the parallelism degree, to start the computation and to get time statistics of last computation.

Among with the implementions the project include a benchmark suite that provide different features for the tests: select which algorithm execute, select the range of input size, select the range for the parallelism degree and many other options.

To compile the project is enough to run the *compile.sh* script inside the project folder. If needed a short help is available for consultation, just run *./bin/benchmark -h*.

5.1 Sequential algorithm

The sequential implementations is simply an iteration over the input vector likely the implementation of the *std::inclusive_scan* function.

As a side note, starting from *C++17* standard the language provide a parallel version of many standard algorithms. In our case it is enough to write:

std::inclusive_scan(std::execution::par_unseq, std::begin(input), std::end(input), std::begin(output);

Unfortunately there is still no support for the parallel algorithms in the *g++* compiler (neither in *clang++*).

5.2 Block-based algorithm

The main problem in the implementation of the block-based algorithm is how to manage, create and assign the blocks to the threads. To solve this issue, in the project we have implemented a class to handle the block division, in particolar the size of a block is $\max(1024, n/p)$: we don't the create blocks of size less than 1024, in order to avoid useless threads that execute only a little portion of the computations.

To implement the parallelism we have used:

- **STL**: for each block spawn a new thread and join them to the main thread at the end of the computations.
- **OpenMP**: the directive *#pragma omp parallel for*, setting the scheduling to *static* and fixing the parallelism degree to *p*.
- **CilkPlus**: the construct *cilk_for* and setting *nworkers = p*.
- **FastFlow**: a *ff::ParallelFor* object with *maxnw = p*, *grainsize* = 1024 and using it to launch in parallel the lambda functions.

5.3 Circuit-based algorithm

In the implementation of this algorithm we use again the block manager together with handler functions to manage the creation of tasks (in particular to generate, given a task number and a iteration number, the operation to execute).

The main problem in this algorithm is that in the execution of the prefix circuit we have assumed to work in-place over the input vector. To solve this issue we have separated the execution of the first task in order to read the data from the input vector and writing the results in the output vector. In this way all the next tasks work in-place over the output vector.

To implement the parallelism we have used the same mechanisms of the block-based algorithm.

Unfortunately, due to task dependency, the implementations have to spawn a large number of threads and this can lead to a non-trascurable overhead in the final completion time.

6 Experimental validation

6.1 experiments details

The experiments were executed on a CPU *Xeon Phi KNC* with 64 physical cores, each of them with a four-way multithreading support (for a total of 256 threads). In particular:

- Each running time is expressed in milliseconds and it is an average of 5 repeated test.
- Each implemented algorithm was tested with $p \in \{2^0, 2^1, \dots, 2^7\}$.
- Each implemented algorithm was tested with $n \in \{2^{20}, 2^{21}, \dots, 2^{29}\}$.
- The correctness of each algorithm has been tested with the sequential algorithm.

In the next subsections we will use T_{seq} to indicate the completion time of the sequential algorithm and $T_{par}(p)$ to indicate the completion time of a generic parallel algorithm with parallelism degree equals to p.

To replicate the experiments just run the benchmark with these parameters:

```
./benchmark –check –all –m-min 20 –m-max 29 –exp 5 –par-max 128
```

6.2 benchmark results

6.2.1 Running time

As first benchmark we simple compare the completion time of each algorithm with each parallelism degree p, using highest size available: $n = 2^{29}$.

		p							
Alg.	Ver.	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
seq	stl	1 565	-	-	-	-	-	-	-
block	cilk	1 560	1 178	623	541	483	343	286	226
block	ff	1 670	1 245	711	399	252	190	201	232
block	omp	1 555	1 169	601	330	179	121	117	117
block	stl	1 571	1 176	609	322	181	124	119	119
circ	cilk	10 631	5 515	2 834	1 717	1 326	845	617	588
circ	ff	10 416	5 718	3 058	1 677	979	644	574	613
circ	omp	10 340	5 362	2 887	1 514	891	576	513	486
circ	stl	10 323	5 415	2 810	1 553	951	661	594	723

What we can see from the first raw data is that, as expected, for $p = 1$ and $p = 2$ the block-based algorithm is similar to the sequential algorithm while, on the other hand, the circuit-based algorithm performe worst than expected but both of them scale very well increasing the parallelism degree.

6.2.2 Speedup

The speedup is calculated as: $s(p) = T_{seq}/T_{par}(p)$ for each algorithm.

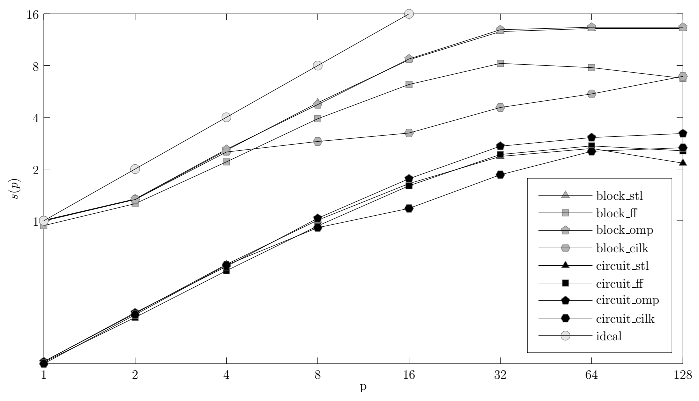


Figure 3: $s(p)$ of each implementations, $n = 2^{29}$

From this plot we can clearly see the differences in the initial speedup between the block-based implementations and the circuit-based implementations. Another thing we can observe is that the slopes of the circuit-based implementations are more regular than the block-based implementations, more similar to ideal and start to slowdown only when $p = 64$.

6.2.3 Scalability

The scalability is calculated as: $scalab(p) = T_{par}(1)/T_{par}(p)$ for each algorithm.

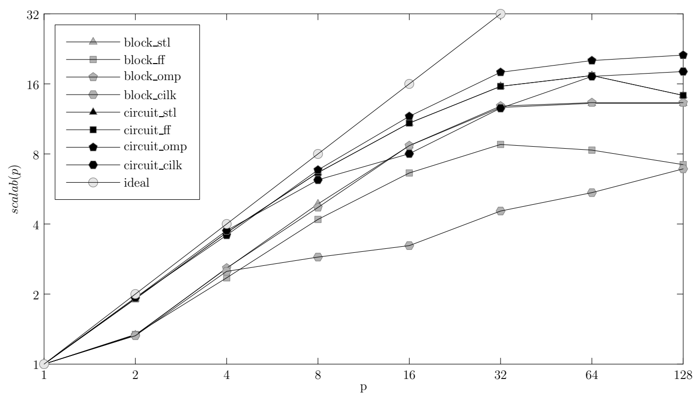


Figure 4: $scalab(p)$ of each implementations, $n = 2^{29}$

From this plot we can see that the circuit-based implementations are slightly better than the block-based implementations but a big outlier is the *block_cilk* that stop to scale very quickly starting from $p = 4$.

6.2.4 Efficiency

The efficiency is calculated as: $\epsilon(p) = T_{seq}/(p \times T_{par}(p))$ for each algorithm.

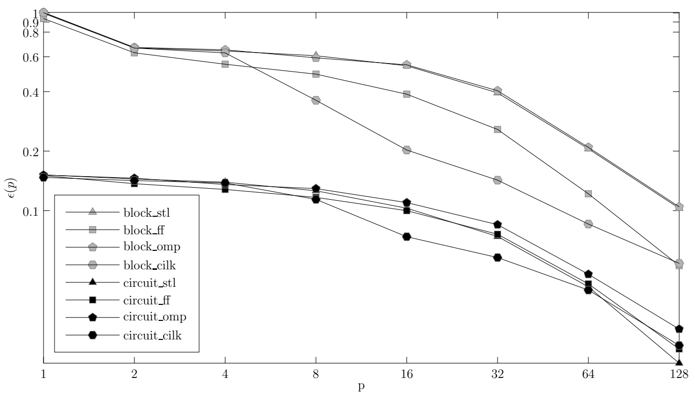


Figure 5: $\epsilon(p)$ of each implementations, $n = 2^{29}$

Once again we can see the differences between the circuit-based implementations and the block-based ones and, as before, we have two outliers that are the two implementations with the *CilkPlus* API.

6.2.5 Completion time

As last test we have compared the running time of each implementations with all the values of p and using all the physical cores.

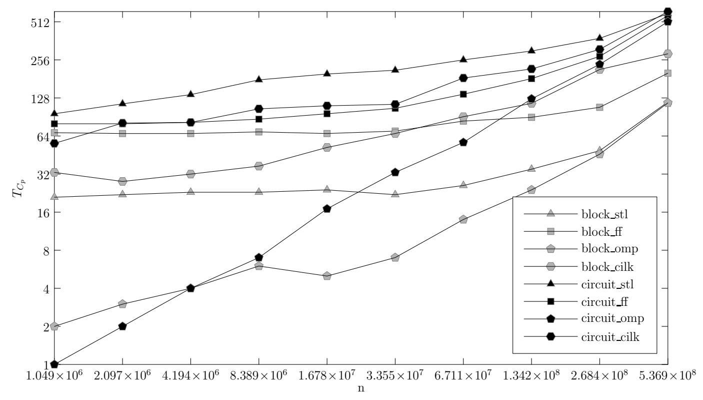


Figure 6: T_{C_p} with $p = 64$ and $p \in \{2^{20}, \ldots, 2^{29}\}$ (lower in better)

From the plot we can see a great slope for the *circuit_cilk* that starts as the lowest implementation from small input but ends to be one of the most efficient with big input.

7 Conclusion

We can divide the observations into two categories: based on the algorithm and on the implementation.

For the algorithms, in our case, the block-based was definitely better than the circuit-based. One of the main causes is definitely the ratio between the size of the input and the parallelism degree, as in the performance modelling we have seen that the circuit-based perform better than the block-based one when $p \simeq n/2$, this is not the case of our experiments as we have $n = 2^{29}$ and $p = 2^7$. On the other hand the circuit-based proves to have a better scalability, so it could be interesting to analyze its behavior with an higher parallelism degree. Also, due its *electronic-like* modelling it could also be interesting to implement the algorithm on the hardware (like FPGA, GPGPU, ...).

Regarding the kind of implementation we have clearly seen that both *circ_cilk* and *block_cilk* have a really poor performance. Surprising is instead the result of *OpenMP* that have the best performance other all the implementations. And finaly the performances of the implementations in *STL* and *FastFlow* are very similar and difficoult to compare. But, from a programmer point-of-view, it was easier to implement the algorithms using the *FastFlow* than in *STL*.